AlphaBorno at BLP-2025 Task 2: Code Generation with Structured Prompts and Execution Feedback

Mohammad Ashfaq Ur Rahman[1], **Muhtasim Ibteda Shochcho**[1], **Md Fahim**[2,3]
[1]Independent University, Bangladesh
[2]Center for Computational & Data Sciences
[3]Penta Global Limited
`{imashfaqfardin, sho25100, fahimcse381}@gmail.com`

Abstract

This paper explores various prompting strategies in the BLP-2025 Shared Task 2, utilizing a pipeline that first translates Bangla problem descriptions into English with GPT-4o, then applies techniques like zero-shot, few-shot, chain of thought, synthetic test case integration, and a self-repair loop. We evaluated four LLMs (GPT-4o, Grok-3, Claude 3.7 Sonnet, and Qwen2.5-Coder 14B). Our findings reveal that while traditional methods like few-shot and chain-of-thought prompting provided inconsistent gains, the integration of explicit unit tests delivered a substantial performance boost across all models. The most effective strategy combined zero-shot prompting with these synthetic tests and a self-repair loop, leading GPT-4o to achieve a top Pass@1 score of 72.2%. These results represent the value of using explicit constraints and iterative feedback in code generation, offering a solid framework that improves the model's code generation capabilities.

1 Introduction

Automatic code generation is a key research area in natural language processing (NLP). This research is driven by benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which evaluate how well systems can turn natural language descriptions into executable programs.

Recent large language models (LLMs) have shown strong results on English-focused benchmarks. However, applying these methods to under-resourced languages like Bangla is challenging (Kabir et al., 2023; Ahmed et al., 2025; Raihan et al., 2025b). Previous studies highlight the difficulties in building effective NLP systems for Bangla (Bhattacharjee et al., 2022; Islam et al., 2021). These challenges arise from limited resources, diverse domains, and ambiguous text forms. To bridge this gap, recent work has introduced multilingual and Bangla-specific resources for code generation. Raihan et al. (2025a) presented mHumanEval, a multilingual extension of HumanEval covering over 200 languages, including Bangla, highlighting performance gaps between high and low-resource languages. Building on this, Raihan et al. (2025b) proposed MBPP-Bangla alongside TigerCoder, a suite of Bangla-focused LLMs, reporting improvements on Bangla code generation tasks.

The Bangla Language Processing (BLP) Workshop 2025 has further advanced this area through a shared task dedicated to Bangla code generation (Raihan et al., 2025c). In this task, participants are required to create Python programs based on Bangla problem descriptions. Evaluation is conducted using a hidden set of unit tests, with systems receiving credit only if their outputs successfully pass all test cases. This setup presents several challenges: first, the problem descriptions may be linguistically ambiguous or domain-specific; second, no reference solutions are available at the time of testing; and third, models must be able to generalize to hidden tests that go beyond the visible examples. These factors make the task a rigorous benchmark for multilingual program synthesis.

To address these challenges, we explore various prompting strategies for LLMs in a shared task context. We created a pipeline that translates Bangla instructions into English with GPT-4 and uses several prompting techniques for code generation. Our study focuses on instruction-only prompting, synthetic test cases, chain-of-thought reasoning, and self-repair loops using execution feedback. We evaluate four models: Qwen2.5-Coder 14B, GPT-4o, Grok-3, and Claude 3.7 Sonnet within these frameworks. Through this comparison, we aim to illuminate how translation and prompting choices affect performance on hidden test cases, providing a practical foundation for future research in multilingual code generation.

2 Background

The Bangla Language Processing (BLP) Workshop 2025 introduced Task 2: Code Generation in Bangla, where the objective is to synthesize Python programs from Bangla natural language instructions (Raihan et al., 2025c). The task follows an execution-based evaluation: systems are credited only if the generated code passes all hidden unit tests. This setting requires models to generalize beyond visible examples and to handle ambiguous or underspecified problem statements, which are common in real-world competitive programming tasks.

For our experiments, we used the official development and test datasets released for the shared task. The development set contains 400 problems, while the test set includes 500 problems. Each problem consists of (i) a Bangla instruction describing the task, (ii) one public test list provided in the form of Python assertions, and (iii) hidden test cases used for leaderboard evaluation. The datasets build upon prior multilingual code generation resources, including mHumanEval (Raihan et al., 2025a) and MBPP-Bangla (Raihan et al., 2025b), which extend HumanEval-style and MBPP-style problems to Bangla.

3 Method

We developed our pipeline using a systematic, staged approach, as illustrated in Figure 1. The process starts with an initial Bangla programming instruction. GPT-4o then performs two tasks in parallel: translating the instruction into English and generating synthetic test cases for edge behaviors. The translated instruction is combined with a selected prompting strategy (e.g., Zero-shot, Few-shot, CoT, or Zero-shot with synthetic tests) and provided to a large language model (LLM) for code generation. The solution is evaluated through run tests, and if it fails, the code enters a self-repair loop for up to three attempts before producing the final code. This design allowed us to effectively assess the contributions of each component.

3.1 Translation Step

We started by running inference on native Bangla instructions using the closed-source model Qwen 2.5-Coder 14B, but performance was limited, with many outputs failing to meet expectations. This confirmed our hypothesis that the model struggled with Bangla programming instructions.

To address this, we introduced a translation step, converting problem instructions to English via GPT-4o, known for its multilingual capabilities. Testing this with Qwen 2.5-Coder 14B revealed improved accuracy: 40.4% for original Bangla instructions versus 46.4% for those translated by GPT-4o. This improvement led us to adopt translation as a standard procedure in our experiments to enhance evaluation reliability and support our structured prompting and self-repair mechanisms.

3.2 Prompting Strategies

We experimented with five prompting setups after establishing translated baselines. Each setup received translated instructions and was evaluated using Pass@1. Below, we illustrate each strategy with a representative problem from the dataset.

Zero-shot. The baseline tests how well the model can generate code from instructions without examples, using a strict format to evaluate its problem-solving ability separately from in-context learning.

Zero-shot Prompt

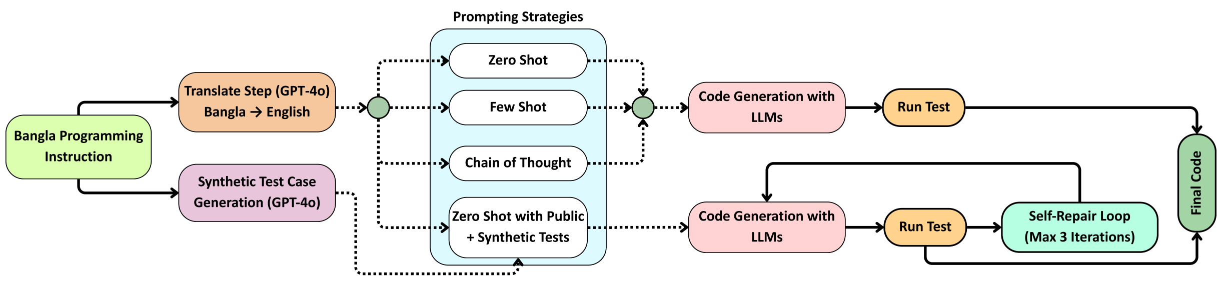
~~~
System prompt:
You are a Python code generator.

STRICT RULES:
- Output ONE fenced code block only:
     ```python ...```
- Inside: exactly ONE function
    definition.
- Function name + args must match
    instruction/tests.
- No helpers/classes unless required
    .
- No comments, explanations, or text
     outside code.
- Only standard library. No I/O.
- Return values only.

User prompt:
Instruction:
Write a function to calculate the
    sum of the digits of each number
    in a given list.
~~~

Few-shot. In this baseline, we provide the model with a few complete task-solution examples after giving it the actual problem statement. This technique tests the model's in-context learning ability, allowing it to infer patterns from the provided demonstrations. Find the prompt in the Appendix A.1

Chain of Thought (CoT). This method prompts the model to follow structured thinking steps before generating code. By explicitly guiding it to



Prompting Strategies
Bangla Programming Instruction
Translate Step (GPT-4o) Bangla → English
Synthetic Test Case Generation (GPT-4o)
Zero Shot
Few Shot
Chain of Thought
Zero Shot with Public + Synthetic Tests
Code Generation with LLMs
Run Test
Code Generation with LLMs
Run Test
Self-Repair Loop (Max 3 Iterations)
Final Code

Figure 1: An overview of the complete code generation pipeline. From an initial Bangla programming instruction, GPT-4o performs English translation while also generating synthetic test cases to capture edge behaviors. The translated instruction is then combined with a selected Prompting Strategy (e.g., Zero-shot, Few-shot, CoT, or Zero-shot with synthetic tests). This combined prompt is fed to an LLM for Code Generation. The resulting code is evaluated by Run Test. If the tests fail, the code enters a Self-Repair Loop (for up to three iterations), where the model attempts to correct its errors. After that, the Final Code is produced.

consider edge cases and constraints, we encourage the model to build a more robust mental plan, leading to a more reliable final implementation. Find the prompt in the Appendix A.2

Zero-shot with public + synthetic tests. This technique provides the model with the problem instruction, with a public test case and a set of synthetic unit tests. Instead of solved examples, these tests act as explicit specifications, guiding the model by defining correct behavior and edge cases.

Zero-shot with Public + Synthetic Tests

```
System prompt:
[Same STRICT RULES as above]

User prompt:
Instruction:
Write a Python function to remove
    the first and last
occurrence of a given character from
     a string.

You must satisfy the following tests
    :

# Public test from dataset
assert remove_Occ("hello", "l") == "
    heo"

# Synthetic edge cases (robustness
    checks)
assert remove_Occ("", "x") == ""
assert remove_Occ("a", "a") == ""
assert remove_Occ("aaaa", "a") == ""
assert remove_Occ("abc", "z") == "
    abc"

Extra requirements:
- Handle empty inputs, negatives,
    duplicates, and large values.
- Preserve ordering if required.
- Ensure deterministic output.
```

```
- Match exact return types (e.g.,
    int vs float, string casing).

OUTPUT:
One fenced Python code block with
    the function only.
```

3.3 Synthetic Hidden Test Generation

To strengthen generalization to the hidden Codabench evaluator, we expanded each problem with an additional set of synthetic test cases. We generated these test cases using GPT-4o, which we prompted to propose inputs that capture edge conditions not covered by the public assertions. We instructed the model to consider empty inputs, extreme numerical values, unusual string structures, type-boundary behaviors, and other corner cases that commonly expose weaknesses in naive implementations. For each problem, GPT-4o produced eight candidate tests. To maintain consistency, we used a standard prompt template when instructing GPT-4o to augment the test set. Here is the prompt we used to generate the test cases:

Test Case Generation Prompt

```
Generate a set of synthetic Python
    assert statements based on a
    provided "function_signature" and
     a list of "public_tests".

Your goal is to create additional
    tests that cover failure-prone
    scenarios not addressed by the
    public tests, including empty/
    null inputs, boundary conditions,
     type mismatches, data-specific
    cases (e.g., unusual strings,
    large numbers, list variations),
    and any semantic corner cases.
```

~~~
CRITICAL: You must not repeat any of
    the provided "public_tests" and
    must return your response only as
    a valid JSON object containing a
    single key, "synthetic_tests",
    which holds an array of strings,
    where each string is a complete,
    runnable Python assert statement.

### INPUT:
function_signature: "def bell_Number
    (n):"
public_tests: "['assert bell_Number
    (2) == 2', 'assert bell_Number(3)
    == 5', 'assert bell_Number(4) ==
    15']"

### EXPECTED OUTPUT FORMAT:
```json
{
  "synthetic_tests": [
    "assert bell_Number(0) == 1",
    "assert bell_Number(1) == 1",
    "assert bell_Number(5) == 52",
    "assert bell_Number(6) == 203",
    "assert bell_Number(10) ==
        115975"
  ]
}
~~~

After the synthetic tests were generated, they were finalized and reused across all models: GPT-4o, Grok-3, Claude 3.7 Sonnet, and Qwen 2.5-Coder. This ensured that every system was evaluated under the same augmented constraints. In practice, these tests made the required behaviors much clearer and improved the models' robustness by providing more explicit signals regarding the expected handling of edge cases.

3.4 Self-repairing Loop

When generated code fails the provided unit tests, we initiate a self-repair loop. The model is provided with the specific AssertionError and prompted to revise its code. This feedback cycle repeats for a maximum of three attempts, allowing the model to iteratively correct its own mistakes based on the test outcomes.

Self-repairing Loop

```
Your previous attempt failed this
    assertion:
AssertionError: assert sum_of_digits
    (0) == 0
Please correct the code while
    keeping the same
function name.
```

3.5 Evaluation Protocol

We report Pass@1, defined as the percentage of problems for which the generated solution passed all asserts. Two levels of evaluation were used:

- **Public asserts** included in the dataset (to check immediate consistency).
- **Hidden tests** on the Codabench leaderboard (to assess generalization).

4 Experiments and Results

We evaluated our pipeline on four models: three closed-source (GPT-4o, Grok-3, Claude 3.7 Sonnet) and one open-source baseline (Qwen2.5-Coder 14B). Our evaluation metric is **Pass@1**, defined as the proportion of problems where the generated solution passed all hidden test cases on the Codabench leaderboard.

4.1 Results & Findings

The results, presented in Table 1, reveal the clear effect of prompt design on model performance. Our analysis highlights several key findings. First, baseline prompting strategies produced inconsistent and often suboptimal outcomes. While Claude 3.7 Sonnet achieved the highest zero-shot score (56.6%), the conventional techniques of few-shot and Chain-of-Thought (CoT) prompting did not guarantee improvements. For instance, few-shot prompting substantially degraded performance for Qwen2.5-Coder (from 46.4% to 41.2%), and CoT underperformed relative to the zero-shot baseline in three out of four models.

Second, the most remarkable performance gain came from providing explicit requirements through synthetic tests. Augmenting the zero-shot prompt with unit tests caused a substantial improvement for all models, with improvements ranging from +13.4 to +19.2 percentage points. This suggests that for code generation, defining concrete behavioral constraints is far more effective than providing abstract examples or reasoning hints.

Finally, the self-repair loop provided an additional, consistent boost to all models, pushing them to their peak performance. This iterative refinement process added a further 1.4 to 3.4 points. In the final configuration, the top-performing models converged, with GPT-4o achieving the highest score at 72.2%, followed closely by Claude 3.7 Sonnet (71.8%) and Grok-3 (71.4%).

Prompting Strategy	Hidden Tests (Pass@1 %)			
	GPT-4o	Grok-3	Claude 3.7 Sonnet	Qwen2.5-Coder
Zero-shot	54.2	52.8	56.6	46.4
Few-shot	56.8	55.4	56.2	41.2
Chain of Thought (CoT)	56.2	53.2	54.6	43.4
Zero-shot + Synthetic Tests	70.2	68.8	70.4	65.6
Zero-shot + Tests + Self-Repair	**72.2**	**71.4**	**71.8**	**69.0**

Table 1: Results comparing prompting strategies across four models. Performance generally increases with more sophisticated prompts, with the combination of explicit synthetic tests and a self-repair loop yielding the strongest results.

4.2 Failure Analysis

The evaluation revealed three main categories of failures: Code Generation Errors, Logical and Algorithmic Flaws, and Source Prompt and Localization Errors. Each category is discussed below.

Code Generation Errors

These failures occurred when the generated code did not conform to the required interface, even though the underlying logic was correct. A common case was when the model produced the right implementation but used a different function name than the one expected by the test case. For example, a function expected to be named `remove_dirty_chars` was instead generated as `str_to_list`, leading to a runtime error because the required entry point could not be found.

Logical and Algorithmic Flaws

Most failures fell into this category. The code executed without error but produced incorrect results due to incomplete algorithms, mishandling of edge cases, or inefficient approaches. As an example, a task required a function to return `-1` when no duplicate element was present in an array. The model's implementation correctly identified duplicates but returned `None` when none existed, causing the output to fail against the expected test condition. For additional failure analysis, please refer to the Appendix A.3.

4.3 Error Analysis

To understand how prompting strategies affected model performance, we analyzed common failure modes and their resolution. Three categories of errors were observed: (i) *interface alignment*, where the logic was correct but mismatched function signatures or formats caused failures; (ii) *algorithmic refinement*, where initial solutions were incomplete and improved once execution feedback clarified the requirements; and (iii) *edge-case handling*, where models overlooked inputs such as negative numbers, empty lists, or formatting constraints.

Zero-shot, few-shot, and CoT prompts often revealed these weaknesses but rarely fixed them. By contrast, adding synthetic test cases made the requirements explicit and guided the models toward outputs that matched the evaluator. Detailed case studies with code transitions are provided in Appendix A.4.

5 Conclusion

This work reveals that a multi-stage pipeline, beginning with machine translation and followed by advanced prompting, is a useful strategy for Bangla code generation. The subsequent use of synthetic tests and self-repair achieved a peak Pass@1 score of 72.2% with GPT-4o. While this approach is effective, considerable challenges remain in overcoming the models' tendency to misinterpret technically precise instructions, even when the code generation instructions themselves are accurate.

Limitations

The pipeline's effectiveness is primarily limited by its reliance on the quality of the initial Bangla-to-English translation, which can introduce errors or lose important nuances. Additionally, there is a more subtle yet noteworthy limitation that goes beyond handling prompts with inherent errors. The failure analysis shows that models often struggle with prompts that are factually correct but include technically specific terms or constraints, which can lead to misinterpretation. This highlights a key challenge in ensuring that models accurately interpret precise instructions.

References

Kawsar Ahmed, Md Osama, Omar Sharif, Eftekhar Hossain, and Mohammed Moshiul Hoque. 2025. Bennumeval: A benchmark to assess llms' numerical reasoning capabilities in bengali. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 17782–17799.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Abhik Bhattacharjee, Tahmid Hasan, Wasi Uddin Ahmad, and Rifat Shahriyar. 2022. Banglanlg and banglat5: Benchmarks and resources for evaluating low-resource natural language generation in bangla. *arXiv preprint arXiv:2205.11081*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Khondoker Ittehadul Islam, Sudipta Kar, Md Saiful Islam, and Mohammad Ruhul Amin. 2021. Sentnob: A dataset for analysing sentiment on noisy bangla texts. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3265–3271.

Mohsinul Kabir, Mohammed Saidul Islam, Md Tahmid Rahman Laskar, Mir Tafseer Nayeem, M Saiful Bari, and Enamul Hoque. 2023. Benllmeval: A comprehensive evaluation into the potentials and pitfalls of large language models on bengali nlp. *arXiv preprint arXiv:2309.13173*.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2025a. mHumanEval - a multilingual benchmark to evaluate large language models for code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11432–11461, Albuquerque, New Mexico. Association for Computational Linguistics.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2025b. Tigercoder: A novel suite of llms for code generation in bangla. *arXiv preprint arXiv:2509.09101*.

Nishat Raihan, Mohammad Anas Jawad, Md Mezbaur Rahman, Noshin Ulfat, Pranav Gupta, Mehrab Mustafy Rahman, Shubhra Kanti Karmakar, and Marcos Zampieri. 2025c. Overview of BLP-2025 task 2: Code generation in bangla. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Association for Computational Linguistics (ACL).

A Appendix

A.1 Few-shot Prompt.

Few-shot Prompt

```
System prompt:
[Same STRICT RULES as above]

User prompt:
Instruction:
Write a Python function to remove
    the first and last
occurrence of a given character from
     a string.

Example 1:
Instruction: Write a function to
    sort a given matrix in ascending
    order.
Solution:
def sort_matrix(matrix):
    [example implementation here]

Example 2:
Instruction: Write a function that
    counts the most common words in a
     list.
Solution:
def most_common_word(words):
    [example implementation here]
```

A.2 Chain of Thought (CoT) Prompt.

CoT Prompt

```
System prompt:
[Same STRICT RULES as above]

User Prompt:
Instruction:
{instruction}

Guidance:
- Before coding, carefully consider
    edge cases (empty input,
    negatives, duplicates, large
    values).
- Make sure return types exactly
    match expectations (int vs float,
     casing in strings, etc.).
- Then output only the final Python
    function in one fenced code block
    .

OUTPUT:
One fenced Python code block with
    exactly one function.
```

A.3 Failure Analysis

Misinterpretation and Missed Constraints

A smaller but critical set of failures was caused by the model's misinterpretation of specific problem requirements, where the prompt was accurate but the model either misunderstood a key technical term or ignored a formatting constraint. For instance, when instructed to generate a function for Woodall numbers (of the form $n \cdot 2^n - 1$), the model produced a correct implementation for the more common Mersenne numbers (of the form $2^n - 1$), effectively missing the critical n multiplier in the formula. In another case, the instruction to "set all odd bits" presented an indexing ambiguity; the model assumed 0-based indexing while the evaluator expected 1-based, leading to failure. Similarly, a task requiring explicit status strings like "Found a match!" failed when the model returned the raw search result, correctly solving the logic but ignoring the strict output format. These cases demonstrate that failures can stem from the model's misinterpretation of terminology or its tendency to overlook precise specifications.

A.4 Error Analysis

This appendix provides detailed examples of how prompting strategies influenced failure to success transitions. We organize the errors into three categories: Interface Alignment Errors, Algorithmic Refinement, and Edge-Case and Output-Type Handling. For each, we show representative cases and the corresponding code transitions, illustrating how synthetic tests corrected baseline shortcomings.

1. Interface Alignment Errors

These tasks failed in the baseline prompts because the function names or output formats did not match what the evaluator expected. Synthetic tests clarified the required interface, which led to corrections.

Example A: Removing digits from strings Zero-shot / Few-shot / CoT: The correct logic was implemented in functions named `remove_digits_from_list`, `remove_digits`, or `remove_digits_from_strings`. The evaluator, however, expected a function named `remove`, so these solutions failed. Synthetic Tests: By making the expected signature explicit, the function was renamed to `remove` while retaining the same logic.

Wrong Code

```
- def remove_digits_from_list(strings
    ):
-     return [''.join(filter(lambda c
    : not c.isdigit(), s)) for s in
    strings]
```

Repaired Code

```
+ def remove(strings):
+     return [''.join(filter(lambda c
    : not c.isdigit(), s)) for s in
    strings]
```

Example B: Converting tuples to strings Zero-shot / Few-shot: Returned comma- or space-separated strings (e.g., `', '.join(map(str, tup))`). CoT: Cast the tuple directly to a string, which preserved parentheses and commas. Synthetic Tests: Required concatenation without separators. The revised version used `''.join(tup1)`, which matched the evaluation format.

Wrong Code

```
- def tup_string(tup1):
-     return ', '.join(map(str, tup1)
    )
- def tup_string(tup1):
-     return ' '.join(map(str, tup1))
- def tup_string(tup1):
-     return str(tup1)
```

Repaired Code

```
+ def tup_string(tup1):
+     return ''.join(tup1)
```

2. Algorithmic Refinement

In these tasks, the baseline prompts produced incomplete or incorrect algorithms. Synthetic tests made the requirements explicit, guiding the model toward correct implementations.

Example A: Top-k frequent elements in nested lists Zero-shot / Few-shot / CoT: Counted only the outer list elements, failing when input contained nested lists. Synthetic Tests: The corrected version flattened the input, used `collections.Counter`, and returned the top-k elements with `heapq.nlargest`.

Wrong Code

```
- def top_k_frequent(nums, k):
-     c = {}
-     for n in nums:
-         c[n] = c.get(n, 0) + 1
-     heap = [(-freq, num) for num,
    freq in c.items()]
-     heapq.heapify(heap)
-     return [heapq.heappop(heap)[1]
    for _ in range(k)]
```

Repaired Code

```
+ def func(nums, k):
+     if k <= 0:
+         return []
+     flattened_nums = [num for
    sublist in nums for num in
    sublist]
+     frequency_counter = Counter(
    flattened_nums)
+     return heapq.nlargest(k,
    frequency_counter.keys(),
+                             key=
    frequency_counter.get)
```

Example B: Splitting a string into a list Zero-shot / Few-shot / CoT: Used `list(string)`, which split into characters. Synthetic Tests: Showed that the intended behavior was splitting into words, corrected with `string.split()`.

Wrong Code

```
- def string_to_list(string):
-     return list(string)
```

Repaired Code

```
+ def string_to_list(string):
+     return string.split()
```

Example C: Sum of amicable numbers Zero-shot: Double-counted amicable pairs. Few-shot: Simplified divisor logic but still risked duplication. CoT: Added a checked set but omitted a boundary condition. Synthetic Tests: Added both a checked set and a guard condition to avoid double counting and ensure correct limits.

Wrong Code

```
- def amicable_numbers_sum(limit):
-     total_sum = 0
-     for num in range(2, limit + 1):
-         divisor_sum =
    sum_of_divisors(num)
-         if divisor_sum != num and
    sum_of_divisors(divisor_sum) ==
    num:
-             total_sum += num
-             if divisor_sum <= limit
    :
-                 total_sum +=
    divisor_sum
-     return total_sum
```

Repaired Code

```
+ def amicable_numbers_sum(limit):
+     total_sum = 0
+     checked = set()
+     for num in range(2, limit + 1):
+         if num not in checked:
+             div_sum =
    sum_of_divisors(num)
+             if div_sum != num and
    div_sum <= limit:
+                 if sum_of_divisors(
    div_sum) == num:
+                     total_sum +=
    num + div_sum
+                     checked.add(num
    )
+                     checked.add(
    div_sum)
+     return total_sum
```

3. Edge-Case and Output-Type Handling

These tasks failed in baseline prompts because of missing sentinel values, inconsistent return types, or overlooked corner cases. Synthetic tests prompted the model to address these issues.

Example A: Decimal to binary Zero-shot / Few-shot / CoT: Returned integers instead of strings and did not handle negative inputs. Synthetic Tests: Corrected by always returning strings and explicitly handling zero and negative values.

Wrong Code

```
- def decimal_To_Binary(N):
-     if N == 0:
-         return "0"
-     binary = ""
-     while N > 0:
-         binary = str(N % 2) +
    binary
-         N = N // 2
-     return int(binary)
```

Repaired Code

```
+ def decimal_To_Binary(N):
+     if N < 0:
+         return "-" +
    decimal_To_Binary(-N)
+     elif N == 0:
```

```
+         return "0"
+     binary = ""
+     while N > 0:
+         binary = str(N % 2) +
    binary
+         N = N // 2
+     return binary
```

Example B: Maximum occurrence Zero-shot / Few-shot / CoT: Returned only the element(s) or `None`. Synthetic Tests: Corrected to return both the element and its frequency.

Wrong Code

```
- def max_occurrences(nums):
-     if not nums:
-         return None
-     count = Counter(nums)
-     max_count = max(count.values())
-     most_common_items = [item for
    item, freq in count.items() if
    freq == max_count]
-     return most_common_items[0] if
    len(most_common_items) == 1 else
    most_common_items
```

Repaired Code

```
+ def max_occurrences(nums):
+     from collections import Counter
+     if not nums:
+         return None, 0
+     count = Counter(nums)
+     max_item = max(count, key=count
    .get)
+     return max_item, count[max_item
    ]
```

Example C: Returning long words Zero-shot / Few-shot / CoT: Treated the input as a list of words, failing when given a raw string. Synthetic Tests: Corrected by splitting the string explicitly.

Wrong Code

```
- def long_words(n, words):
-     return [word for word in words
    if len(word) > n]
```

Repaired Code

```
+ def long_words(n, words):
+     return [word for word in words.
    split() if len(word) > n]
```